Circuit Design Tool User's Manual[1]

Keith M. Miyake
Donald E. Smith

LCSR-TR-191 Revision 2

Laboratory for Computer Science Research
Computing Research and Education Building
Busch Campus, Rutgers University
New Brunswick, New Jersey 08903

October 1992

[1]This work was supported by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668.

Contents

Chapter 1

Introduction

The CAM chip design has been produced in a UNIX software environment using a design tool that supports definition of digital electronic modules, composition of these modules into higher level circuits, and event-driven simulation of these circuits. Our design tool provides an interface whose goals include straightforward but flexible primitive module definition and circuit composition, efficient simulation, and a debugging environment that facilitates design verification and alteration.

The tool provides a set of primitive modules which can be composed into higher level circuits. Each module is a C-language subroutine that uses a set of interface protocols understood by the design tool. Primitives can be altered simply by *recoding* their C-code image; in addition new primitives can be added allowing higher level circuits to be described in C-code rather than as a composition of primitive modules - this feature can greatly enhance the speed of simulation[1].

Effective composition of primitive modules into higher level circuits is essential to our design task. Not only are the *standard* features of a description language required but in addition, features such as recursive descriptions of circuit composition, parameterized module descriptions, and strongly-typed port types are essential to efficient circuit design. These features are supported by our design tool's composition language which allows the user to specify a hardware description in a C-like syntax. Parameterized modules, recursive and iterative descriptions, macro-like capability to describe collections of wires (i.e., cables), and decision making support that allows context sensitive module expansion are provided by our tool. In addition, our tool can determine the cost of a circuit based on the costs of its primitive modules. This feature is not *exact* but does provide a good approximation of the complexity of the designed circuit.

Simulation is performed by an event-driven simulator that handles gates as well as tri-state bi-directional busses and provides the user not only with a view of what a circuit is computing but also control over the circuit so that *design flaws* can be effectively isolated

[1]Converting a higher-level circuit into a primitive module is straightforward when the timing of the primitive module need not be identical to the higher level circuit. Higher level circuits can be converted to primitive modules with identical time performance; however, the conversion process is much more complex.

and corrected. The simulator is controlled with a command language which allows the user to *see* a wire or set of wires, as well as change the values on wires. Operations can be done *immediately* (i.e., at the time the user enters them) or scheduled to take place at a specified time. Simulations can be run for a specific period of time or until a certain condition is detected in the hardware. They can be controlled from the keyboard or indirectly from a file.

The design tool consists of two main parts: the command and definition languages. The *definition language* is used to read circuit definitions. The *command language* controls the actions of the simulator. These topics are detailed in sections 2.5 and 2.6.

Following is a brief introduction on the definition and creation of circuit models. It defines many terms used later.

1.1 Circuit Definition

The circuit definition language describes connections between primitive objects. These objects, called *primitive modules*, have functionality predefined in the design tool. Primitive modules have a special set of entry points which are connected when forming the circuit model. These entry points are called *ports* and the connections between them are referred to as *signals* or *wires*. There is a causality between connected modules. Execution of a module may affect modules connected to it.

The design tool reads descriptions using a definition language. The language consists of two types of object definitions: module and cable. *Cable definitions* group related signals together. *Module definitions* specify primitive modules and their connections. A module may define other modules as children of itself, and specify connections between its child modules. In this case the module is referred to as a *composite module.*

The circuit is built from a set of hierarchical module and cable definitions. Flattening the hierarchy produces the basic model of a set of primitive modules connected by wires.

In order to name objects in the hierarchical design, *hierarchical names* are used by the design tool. These names specify objects which cannot be directly referenced within the current context. This is done by supplying a list of names specifying a path to the object. Each field in the composite name is separated by the dot character '.'.

1.2 Model Creation

The creation of a circuit model is performed in phases.

When a cable or module definition is read, its syntax is checked and the definition is stored as a *master definition.* These definitions may have input arguments which need assignment.

When a module is created, input arguments to master definitions are assigned resulting in

a new definition type. These definitions, which have specific input arguments, are referred to as *definition instances.* Each definition instance is fully examined, checking the consistency of the connections made and the referenced modules.

The circuit model is made from these definition instances. The model is designed for speed in simulating the functionality of the circuit and contains all structures necessary for simulation. Such a model is called a *generated module* or *simulation instance.*

Chapter 2

The Definition Language

The definition language is used to describe digital electronic circuits by building hierarchical structures connecting primitive modules. A definition file consists of a sequence of module and cable definitions. Modules come in two types: primitive and composite.

Primitive modules are the basic building blocks of the definition language. These objects perform operations defined by C functions which have been precompiled into the design tool. A list of primitive modules is given in appendix B.

A composite module definition defines submodules of itself and connections to be made among their ports, as well as its own ports. Attaching submodule ports causes interactions between the operations of the respective modules.

Cable definitions allow signals to be identified in groups, which simplifies connection of ports.

Module and cable definitions are similar in structure and are analogous to functions in a conventional programming language. They may have formal input arguments and may use other definitions (as well as their own) recursively. Termination of such recursion is not assured.

2.1 Syntax Conventions

The language syntax descriptions used in this manual is a variant of the Backus-Naur form.

Following is a list of syntax rules:

1. **Boldface** type denotes reserved words.
2. Lowercase words, which may have embedded underscores, denote syntactic constructs.
3. Character tokens are shown using `typewriter` type. Most punctuation characters are used as character tokens, with exceptions stated below. Note that the exceptions are printed in Roman type.

4. The vertical bar '|' separates alternate syntax items when it is used at the beginning of a line.

5. Square brackets ('[', ']') enclose optional items.

6. The dollar sign '$' in a syntax rule denotes the remainder of the line as a comment. '$' is not used in the syntax.

2.2 Variables and Assignment

A variable is a name associated with an integer value in a module or cable definition. Variables have no meaning outside the current definition. A variable name may be any valid string token (quoted or unquoted; see appendix A). There are no arrays of variables. A variable may have the same name as signals, modules, or cables since its context is distinct. There are three variable types: input, loop, and assignment. Within a specific definition, a variable may be used as only one type.

Input Variables

Input variables are arguments to a module or cable definition. They are determined at invocation and may not be reassigned within the current object. These variables are valid throughout the current object. Each input variable of a definition must be given a value upon use.

Loop Variables

Loop variables are used in **for** loops in the component section of modules. Each **for** loop controls the assignment of a single loop variable. Loop variables are only valid within the controlling loop, and may not be reassigned within the loop.

Assignment Variables

Assignment variables are used in the component section of modules. They are set using the **assign** statement (string_token <- arith_expr ;). This assigns the current value of the expression to the variable. Once a variable has been assigned to, it is valid until the end of the module. Each subsequent use of the variable gets the assignment value unless the variable has been reassigned. Assignment variables may not be reused as loop variables.

Control flow variations resulting from **if** statements or loops may allow an assignment variable to be referenced prior to assignment.

Cables only have input variables since they have no component section.

2.3 Expressions

Expressions are used in various ways to control the assembly of modules. There are two types of expressions: arithmetic and logical. *Arithmetic expressions* result in integer values. *Logical expressions* return one of the values TRUE or FALSE. Arithmetic and logical expressions are not interchangeable.

Arithmetic Expressions

Arithmetic expressions compute integer values. They may be string tokens (variables), numeric tokens (constants), or may be created by application of an arithmetic operator to one or more arithmetic expressions.

```
arith_expr :=
        string_token                    $ variable value
      | numeric_token                   $ constant value
      | - arith_expr                    $ arithmetic negation
      | ( arith_expr )                  $ arithmetic grouping
      | arith_expr * arith_expr         $ multiplication
      | arith_expr / arith_expr         $ division
      | arith_expr % arith_expr         $ modulus
      | arith_expr + arith_expr         $ addition
      | arith_expr - arith_expr         $ subtraction
```

Division operations return a truncated result (using C convention), since integer division is not exact.

There are three levels of arithmetic operator precedence. Unary operators (negation and grouping) share the highest precedence. Multiplication, division, and modulus (*, /, %) have equal precedence, below that of the unary operators. Addition and subtraction (+, -) share the lowest precedence.

All binary arithmetic operators associate left-to-right.

Logical Expressions

Logical expressions compute the value TRUE or FALSE. They are constructed by the use of relational or logical operators. *Relational operators* produce a logical expression based on the validity of a relational query between two arithmetic expressions. *Logical operators* use one or two logical expressions to produce a single logical expression. There are no logical variables or constants.

```
log_expr :=
        arith_expr > arith_expr          $ greater than
      | arith_expr >= arith_expr         $ greater than or equal to
      | arith_expr < arith_expr          $ less than
      | arith_expr <= arith_expr         $ less than or equal to
      | arith_expr = arith_expr          $ equal to
      | arith_expr ! arith_expr          $ not equal to
      | ~ log_expr                       $ logical negation
      | { log_expr }                     $ logical grouping
      | log_expr & log_expr              $ logical AND
      | log_expr | log_expr              $ logical OR
```

Note that the vertical bar in the logical OR represents the character '|'.

The use of arithmetic expressions as operands eliminates precedence or associativity with regard to relational operators.

There are three levels of logical operator precedence. Unary logical operators (negation and grouping) have the highest precedence, followed by logical AND. Logical OR has the lowest precedence of logical operators.

Logical grouping syntax is distinct from that of arithmetic grouping. This reinforces the idea of noncompatibility between expression types.

2.4 Naming Conventions

Each child object (signal, cable, or submodule) in a definition must be given a unique name. This allows unambiguous signal naming within simulation instances (for design verification). Names of child objects must be string tokens (quoted or unquoted).

An object name may have a single associated array index. This index is specified by an arithmetic expression enclosed in square brackets following the name. The string token, excluding the array index, is called the *root name* of the object.

```
object_name :=
        string_token
      | string_token [ arith_expr ]
```

Note that the square brackets do not represent optional arguments.

Example:

The root name of an object "a[5]" is simply "a".

It is often useful to name lists of objects. In this case, a modified array notation, called an object list, may be used to specify a range of array indices. This is done by supplying a start and end index for the array, separated by a colon ':'. The notation is equivalent to supplying each object name in order, beginning with the start index, and iterating until the

end index is reached. If the start index is less than the end index iteration increments by one, otherwise it decrements by one.

object_list :=
    string_token [arith_expr : arith_expr]

Note that the square brackets do not represent optional arguments.

Example:
"a[1:2]" expands to "a[1]" "a[2]".
"a[2:1]" expands to "a[2]" "a[1]".

A list of objects may contain single object names and object lists.

object_name_list :=
    object_name
    | object_list
    | object_name object_name_list
    | object_list object_name_list

In a module definition, each internal subcomponent or signal must have a distinct name (root name and index). Also, objects with the same root name must have similar types. This means that signals, components and cables may not share root names. Furthermore, components or cables which share a root name must share the same master definition. These checks are performed during the creation of module definition instances.

It is often necessary to name an object which cannot be directly referenced from the current level. In this case a composite name is used, using the dot character '.' to separate levels. This is referred to as a hierarchical name.

hierarchical_name :=
    object_name
    | object_name . hierarchical_name

Example:
The hierarchical name "a.b" refers to an object "b" which is a child of object "a", where "a" is a child of the current module.

Hierarchical naming may be used with array expansion, in which case rightmost indices are expanded first.

hierarchical_list :=
    object_name
    | object_list
    | object_name . hierarchical_list
    | object_list . hierarchical_list

Example:
"a[1:2].b[3:4]" expands to "a[1].b[3]" "a[1].b[4]" "a[2].b[3]" "a[2].b[4]".

The hierarchical analog to an object name list, called a hierarchical list, may now be defined. Note that every hierarchical name is also a hierarchical list.

hierarchical_name_list :=
    hierarchical_list
    | hierarchical_list hierarchical_name_list

2.5 Cable Definitions

A cable represents an ordered list of signals, each signal having an associated type. Signal typing is used to ensure that the use of a module is consistent with its definition.

cable_definition :=
    cable string_token [(variable_list)] typed_signal_list **end**

variable_list :=
    string_token
    | string_token , variable_list

typed_signal_list :=
    signal_name_list signal_type
    | signal_name_list signal_type typed_signal_list

signal_name_list :=
    object_name_list
    | cable_use
    | object_name_list signal_name_list
    | cable_use signal_name_list

signal_type :=
    input
    | **output**
    | **inout**

The string token following **cable** is the cable name. This name is used for future references to the cable. The variable list is a list of input variables for the cable. When the cable is used, each input variable must be given a value. The typed signal list is a list of the wires which comprise the cable. It may include cables uses, which is defined below. Each signal is given one of three allowable types: **input**, **output**, or **inout**. The meaning of the types will be described in section 2.6.

Example:

```
cable c1
   s1 s2 input
   s3 output
   s4 inout
end
```

In the example, signals "s1" and "s2" are both **input**.

After a cable has been defined, it may be used anywhere that a signal may be used. This includes being used in other cable definitions. The following syntax defines a name as a use of a cable.

cable_use :=
    cable string_token [(argument_list)] object_name
    | **cable** string_token [(argument_list)] { object_name_list }

argument_list :=
    arith_expr
    | arith_expr , argument_list

The string token following **cable** is the name of a cable definition. The argument list given must be exactly the same size as the number of input variables to the cable definition. Following the arguments is the list of new cable instance names.

Example:

```
cable c1 ci1
cable c1 { ci2 ci3 }
```

The first use defines a single instance "ci1" of cable "c1". The second use defines two additional instances, "ci2" and "ci3", of cable "c1".

When a cable is used in another cable definition, the type of the resultant signal depends on both the signal type given in the previous definition, and the type given to the cable use.

The following matrix shows the retyping rules:

cable type	*subsignal type*		
	input	**output**	**inout**
input	input	output	inout
output	output	input	inout
inout	inout	inout	inout

After a cable instance has been defined, each use of the instance name represents the list of its component signals in order. Each signal name in the list is a hierarchical name consisting of the cable instance name and the component signal name. Individual signals within the cable may be accessed by naming the signals hierarchically.

Example:

```
cable c2
   s1 input
   s2 output
end
```

```
cable c3
   cable c2 sc1 input
   cable c2 sc2 output
   cable c2 sc3 inout
end
```

In cable "c3", signal "sc1.s1" would be **input** and "sc1.s2" would be **output**. Because of retyping, signal "sc2.s1" would be **output** while "sc2.s2" would be **input**. Both subsignals of "sc3" would be **inout**.

Example:

If we make a instance "ci4" of the cable type "c1", individual signals may be referenced as "ci4.s1", "ci4.s2", "ci4.s3", and "ci4.s4". This set of signals, in order, can be referenced simply as "ci4".

Cable definitions may use other cable definitions, including those which are not yet defined (forward referencing). There is no check for recursive cable references, which do not terminate.

2.6 Module Definitions

Two types of modules (primitive and composite) are used in circuit designs. Primitive modules are objects with predefined functions. Composite modules define connections between primitive and composite modules.

module_definition :=
module string_token [(variable_list)] [cost_section]
[port_section] [signal_section] [component_section] **end**

The string token following **module** is the module name. This name is used to reference the module in future use. As with cable definitions, when a module is used each input variable must be given a value.

Additional module examples are given in appendix C.

Cost section

The cost section is used to estimate the relative expense of building modules using several technologies. Each module definition instance has associated cost values. These costs may be explicitly defined in the cost section, or may be implicitly defined as the sums of the costs of its submodules. Primitive modules should define explicit costs with a cost section.

Composite modules should include a cost section if the hardware implementation of the module does not correspond to the functional model represented by its subcomponents.

cost_section :=
    costs cost_pair_list

cost_pair_list :=
    cost_pair
    | cost_pair cost_pair_list

cost_pair :=
    nmos : arith_expr
    | **cmos** : arith_expr
    | **gateInput** : arith_expr

Currently there are three cost criteria: **nmos**, **cmos**, and **gateInput**. If a technology cost is given more than once, the last cost pair is used.

Example:

```
module m1(v1)
   costs
      nmos:  2*v1
      cmos:  10
      gateInput:  20
end
```

Port section

The port section is an ordered list of the external connections of the current module. Ports are special signals which are used to connect to the module in later uses. A module with no ports cannot be referenced by another module. The order of port signals is important and determines proper connection of the module.

port_section :=
    ports typed_signal_list

The typed signal list is the same as used in cable definitions, with the same subsignal retyping rules.

Type information defines the proper use of the signal in the module and what connections are allowed if the module is referenced by a composite module.

input implies that the signal is generated from an external source.

output implies that the signal is generated within the current module.

inout does not state the source of the signal. It causes the signal to be a (bi-directional) bus, which must be driven by tri-state drivers.

The following rules govern valid connections to each type of port signal within the current module:

input: No output signal may be connected to the signal. At least one primitive descendent module must use the signal as an input.

output: At least one primitive descendent module must use the signal as an output.

inout: At least one primitive descendent module must use the signal as an input or output. Additionally, every connected output must be a a tri-state driver (the signal is a bus).

Missing or inconsistently typed signal connections are reported upon creation of module definition instances.

Example:

```
module m2
   ports
      p1 input
      p2 output
      p3 inout
end
```

Signal section

The signal section defines internal signals of the current module. These internal signals must be distinct from port signals and may not be referenced by other modules. Every signal used in a module definition must be defined in either the port or signal section. The order in which internal signals are defined is not important.

signal_section :=
signal signal_name_list

Each signal defined in the signal section is given a special type of **internal**. If a cable use is defined in this section, all resulting signals are also typed as **internal**.

The **internal** type means that the signal is both generated and used by primitive descendents of the current module.

Example:

```
module m3
   signals
      s1 s2 s3
end
```

Component Section

The component section determines how composite module are built from other modules. This is accomplished by 'executing' component statements in order, similar to conventional programming languages. Primitive modules, whose functions are defined by C code, do not use their component sections.

component_section :=
    components component_statement_list

component_statement_list :=
    component_statement
    | component_stmt component_statement_list

component_statement :=

submodule_statement	$ declare a child module
| assign_statement	$ assign a value to a variable
| join_statement	$ create a link between a group of signals
| error_statement	$ print an error message
| grouping_statement	$ group multiple statements
| if_statement	$ execute statements conditionally
| for_statement	$ execute a statement loop iteratively
| while_statement	$ execute a statement loop conditionally
| break_statement	$ exit from loops

Submodule Statement

Submodule declares a module as a child of the current module. It also designates attachment of signals to the ports of the child module.

submodule_statement :=
    object_name string_token [(argument_list)] hierarchical_name_list ;

The initial object name is the local name given to the submodule. This name is used to refer to the child module within the current module. Specifically, it is used in hierarchical naming. The next string token is the name of a module definition. The number of arguments given must match the number of input variables of the module definition. Next is a list of signals to be attached to the ports of the child module. Because every signal must be declared in the port or signal section, references to cables use hierarchical names (and not cable uses). Each signal in this list will be connected to the corresponding port of the previously defined module in order. The signal list must be the same size as the number of ports of the previously defined module. The port and connecting signal must conform, using the rules stated under the port section.

Example:

```
module m4
   ports
```

```
      m4i input
      m4o output
   signals
      ic1
end

module m5
   ports
      m5i input
      m5o output
   components
      sc1 m4 m5i m5o;
end
```

In the example, module "m5" defines a child module of type "m4" and gives it the local name "sc1". The hierarchical name which refers to the signal "ic1" in "m4" is "m5.sc1.ic1". Note that the ports of "m4" and the connecting signals in "m5" correspond in type.

Assign Statement

Assign associates an integer value with a variable. The target of **assign** may be any unused variable name, or an assignment variable. Execution of **assign** causes the expression value to be computed and assigned to the variable.

assign_statement :=
    string_token <- arith_expr ;

The string token names the variable to be assigned.

Example:

```
module m6
   components
      v1 <- 2;
      v2 <- v1 * 2;
      v1 <- 1;
end
```

The first **assign** creates a new variable "v1" with a value of 2. The second creates "v2" and uses "v1" to compute "v2" as 2*2 = 4. The final **assign** changes "v1" to 1, but does not affect "v2".

Join Statement

Join merges a set of signals to form a single signal. After a **join** has been completed, any

member can be used to represent the set in other component statements (including joins).

Every signal used in a **join** must be declared in the port or signal section of the module.

The merging of signals caused by a **join** may introduce non-obvious inconsistencies in the connection of modules. These inconsistencies are reported upon execution of the **join**.

join_statement :=
join [hierarchical_name_list] ;

Note that the square brackets above do not indicate an optional argument.

Example:

```
module m7
   signals
      s1 s2 s3
   components
      join [s1 s2];
      join [s2 s3];
end
```

The first **join** merges signals "s1" and "s2". The second merges the signal "s3" with the signal which is the **join** of "s1" and "s2".

Error Statement

Error allows the user to print a message during the course of module generation. The message is a single string (no variables), and is designed mainly for identifying situations that should not occur.

error_statement :=
error string_token ;

Execution of **error** causes activation of an error message with the error flag mask activated. The error mask value is given in appendix D. These messages may be suppressed or may cause program termination by options in the **simrc** file.

Grouping Statement

Grouping allows multiple component statements to act as a single statement lexically. This allows multiple statements to be used as targets in **if**, **for** and **while** statements. **Grouping** has no affect in other contexts.

grouping_statement :=
{ component_statement_list }

Note that there is no semicolon following the grouping statement.

Grouping does not affect the lexical scope of any variable.

If Statement

If allows conditional execution of a statement depending on the result of a logical expression. Multiple statements may be executed by the use of **grouping**.

if_statement :=
    if log_expr component_statement [**else** component_statement] ;

If executes the first component statement when the logical expression is TRUE. When the logical expression is FALSE, the second component statement (in the optional **else** clause) is executed if available.

For Statement

For executes a statement a specified number of times. Multiple statements may be executed by the use of **grouping**.

For evaluates two bounding expressions once to find the inclusive range for its loop control varable. The target statement is then repeatedly executed with the loop variable set to each value in the range. The loop variable is initially set to the value of the first expression. If the first expression is less than the second expression, the loop variable is incremented by one after each iteration; otherwise the variable is decremented by one.

The loop variable is not allowed to be a input or an assignment variable, and may not be assigned within the loop. This guarantees termination of **for**.

for_statement :=
    for string_token = arith_expr , arith_expr component_statement

While Statement

While executes a statement as long as a logical expression remains TRUE. Multiple statements may be executed by the use of **grouping**.

While first evaluates the controlling logical expression. If it is TRUE, the target statement is executed, and **while** is reexecuted. If it is FALSE, execution continues at the statement immediately following the **while**.

Termination of **while** is not guaranteed. There is no check for non-termination.

while_statement :=
    while log_expr component_statement

Break Statement

Break is used to halt processing of **for** and **while** statements. **Break** disregards pending statements in the current target component, and continues execution at the statement

immediately following the current **for** or **while** statement.

Break takes an argument which is the number of nested loops to break. A nonpositive argument has no effect. If the argument is larger than the number of nested loops, creation of the module is completed at the **break**. **Break** does not affect parent modules.

break_statement :=
    break arith_expr ;

Chapter 3

Command Syntax

The command syntax controls what actions are taken. These commands control the definition and execution of circuit models.

Commands are normally read from standard input. They may be directed from a file by using an input flag or a command statement.

3.1 Filenames

A special syntax is accepted to facilitate the use of filenames. Filenames are allowed to be string tokens separated by periods '.'. This allows specification of most local filenames without having to use quoted strings.

file_name :=
    string_token
    | string_token . file_name

Quoted strings must be used in order to use the UNIX directory structure.

3.2 Current Generated Module

The name of the last generated module to be referenced is saved. This is known as the *current generated module.* The current generated module is used when commands are issued which omit the optional module name. The current generated module is automatically set by **generate**, and may be changed using **set**.

3.3 Current Submodule

Each generated module has a single *current submodule*. The current submodule is used as a shorthand notation for a single submodule in each generated module. This allows simple reference to the submodule during testing.

The current submodule is referenced by beginning a command name with character '@'.

The current submodule of a simulation instance is initially the top level module generated. It may be changed using **set**.

3.4 Parent Constructor

The command naming syntax contains a parent constructor '^'. As each field is read in the left-to-right expansion of a hierarchical name, the partial name corresponds to an object in the current module. When the parent constructor is read, the new object referenced by the partial name is set to the parent of the current object.

The parent constructor is usually used in conjunction with the current submodule '@'.

Use of the parent constructor '^' with an array of child modules may cause problems.

3.5 Command Naming

Names in the command syntax are similar to hierarchical names in definitions. There are additional rules which apply to command names:

- The name of the current generated module or the current submodule identifier '@' must be the first field in the hierarchical name.
- There is a *parent constructor* '^' which changes the target to the parent of the current target.

We now define an object name in the command syntax.

command_object :=
    @
    | string_token
    | @ . command_object_tail
    | string_token . command_object_tail

command_object_tail :=

^

| object_name

| ^ . command_object_tail

| object_name . command_object_tail

We also define a list of command objects defined by array expansion. This corresponds to the hierarchical list in the definition syntax.

command_list :=

@

| string_token

| @ . command_list_tail

| string_token . command_list_tail

command_list_tail :=

^

| object_name

| object_list

| ^ . command_list_tail

| object_name . command_list_tail

| object_list . command_list_tail

Finally, a general list of command names is defined. Note that every command object is also a command list.

command_object_list :=

command_list

| command_list command_object_list

3.6 Design Tool Commands

```
command_statement :=
        source_statement                $ read in a definition file
      | read_statement                  $ read commands from a file
      | close_statement                 $ close an open command file
      | pause_statement                 $ transfer control
      | repeat_statement                $ loop read a command file
      | generate_statement              $ generate a module for simulation
      | run_statement                   $ simulate a generated module
      | reset_statement                 $ reset signals in a generated module
      | save_statement                  $ save current simulation state to a file
      | load_statement                  $ load simulation state from a file
      | destroy_statement               $ destroy a generated module
      | clear_statement                 $ clear all current definitions
      | set_statement                   $ set options
      | assignment_statement            $ assign values to signals
      | trap_statement                  $ set conditions to halt simulation
      | untrap_statement                $ remove halting conditions
      | show_statement                  $ display signal vectors
      | showvector_statement            $ display signal vectors as a group
      | showmessage_statement           $ display a message
      | showtime_statement              $ display the current simulation time
      | add_statement                   $ add signals to a display list
      | showlist_statement              $ display signals in the display list
      | display_statement               $ enable printing of changed signals
      | undisplay_statement             $ disable signal printing
      | timed_statement                 $ execute a command during simulation
      | quit_statement                  $ exit the program
```

Source Statement

Source reads a file of module definitions. The entire file is read using the stated definition language rules. Module syntax is checked as the definition file is read. Definitions are checked for consistency only when referenced by a **generate**.

```
source_statement :=
        source file_name ;
```

If **source** causes redefinition of a module, the new definition will be used for future *module definition instances* (not simulation instances). Previously defined instances will continue to use the old definition. Module redefinition not recommended.

Read Statement

Read causes commands to be read from a file. The file is read until the end-of-file is reached, or a **pause** is executed in the file. Commands are again read from the current source following exit from the named file.

read_statement :=
    read [file_name] ;

If the filename is omitted, the last open file read is used.

The file is closed after a **read** if the entire file has been read. If the named file is already open, **read** continues at the current position in the file.

Close Statement

Close closes a file left open by a previous **read**. This allows a file containing a **pause** to be reread from the beginning of the file.

close_statement :=
    close [file_name] ;

If the filename is omitted, the last open file read is closed.

Pause Statement

Pause stops reading of the current source of command input. Command input is then read from the previous source.

pause_statement :=
    pause ;

A **Pause** in a command file causes reading of the file to stop. The file is kept open, and a subsequent **read** will continue at the command following the **pause**. In the interactive (top) level, **pause** exits the design tool.

Repeat Statement

Repeat causes repetitive reading of a command file until a test passes. It reads a signal value as a test for completion once each time the file is read.

There are two versions of **repeat**: **while** and **until**.

While checks the signal before reading the file, Execution continues as long as the signal value is logical high.

Until reads the file before checking the test signal. It continues execution as long as the value is **not** logical high. **Until** always reads the file at least once.

Note that the two versions use inverse testing conditions.

repeat_statement :=
    repeat file_name **while** command_signal ; $ check before loop
    | **repeat** file_name **until** command_signal ; $ check after loop

If multiple tests are needed for the halting condition, the circuit must be designed in hardware.

There is no check for termination of **repeat** statements.

Calling **repeat** on files which contain the **pause** command should be done with caution.

Generate Statement

Generate creates a simulation instance of a module definition. The module should have been previously read using a **source**. The simulation model is used to test correctness of designs. Generating a module causes all consistency checks on submodule use to be performed. The consistency rules have been stated along with the definition syntax.

Each generated module is set to a special initial state in which all non-constant signals are set to the undefined value.

Gereration of a module causes it to become the current generated module.

generate_statement :=
    generate string_token [(argument_list)] ;

The string token specifies the name of the module to be generated. The module name will be used to reference the simulation instance. There is no way to distinguish between two instances of the same module. The argument list is a list of integers which must correspond to the input variables of the module.

Run Statement

Run executes a simulation run of a generated module. Events queued for the module are evaluated until all events have been processed or a halt command has been issued. **Run** is detailed in section 4.

run_statement :=
    run [string_token] ;

The string token specifies the generated module to be run. If omitted, the current generated module is run.

A simulation run may be aborted by an interrupt signal (control-C). Such an interrupt sets command input to the interactive level, or exits the design tool if it is being run in batch mode. Aborting a simulation run does not affect pending events.

Reset Statement

Reset causes a generated module to be set to its special initial state. Pending events are removed from the event queue, then non-constant signals are set to the unknown value.

reset_statement :=
    reset [string_token] ;

The string token specifies the generated module to be reset. If omitted, the current generated module is reset.

Save Statement

Save saves the state of the current generated module to a file. A new file with the given filename is created and the current state is saved. The save file consists of ordered listings of signal values, but does not specify corresponding signal names.

save_statement :=
    save [file_name] ;

Load Statement

Load sets the state of the current generated module using a file previously created using **save**. The current generated module must be the same type as the saved module.

load_statement :=
    load [file_name] ;

The only verification of the module type is that the length and number of signal value lists are correct.

Destroy Statement

Destroy frees a generated module which is no longer needed. Destroying a module does not affect any other generated modules or any definitions.

destroy_statement :=
    destroy [string_token] ;

The string token specifies the generated module to be destroyed. If omitted, the current generated module is destroyed.

If the current generated module is destroyed, it will be ill-defined until reset by a **set** or **generate**.

Clear Statement

Clear deletes all definitions and simulation instances. This is equivalent to restarting the design tool.

clear_statement :=
    clear ;

Set Statement

Set is used to change settings in the design tool. Things changed by the set command include the current generated module, the current submodule (of the current generated module), and flags for signal display and output verbosity.

- The current generated module is used when the module name is not specified in a command.
- The current submodule is used as the initial path object in command names which use '@' as the initial field.
- The signal display flag is set using **display** and reset using **undisplay**. When the display flag is set, each signal prints its state whenever it changes value. The flag is initially reset.
- Output verbosity is set using **verbose** and reset using **brief**. Verbosity controls the amount of information printed as commands are executed. The flag is initially set.

set_statement :=
    set simulation string_token ;
    | **set @** command_object ;
    | **set display** ;
    | **set undisplay** ;
    | **set brief** ;
    | **set verbose** ;

In the first variation, the string token refers to a simulation instance. This instance becomes the current generated module.

In the second, the new current submodule '@' is specified by the command object. The command object must be a module, not a signal or cable. The previous value of '@' may be used to specify the new object.

The flag set by the **display** option is independent of those used by the **display** statement.

Assignment Statement

Assignment sets a signal value in the current generated module. Assignment events are

put into the event queue of the current generated module. These events are completed on the next **run** of the module.

assignment_statement :=
    command_signal_list <- numeric_token ;

The command signal list has been described under command naming.

The numeric token may be a value constant, or a binary, octal, hexadecimal, or decimal number.

A value constant may be one of keywords from the following list. In this case, all signals in the list are assigned the same value, which is the value corresponding to the keyword.

value_constant :=

LSIG	$ logical low
| **HSIG**	$ logical high
| **TSIG**	$ tri-state value
| **USIG**	$ undefined
| **XSIG**	$ bad signal value

Numbers are converted to a binary representation then assigned in order with the least significant bit assigned to to the rightmost signal. A 0 bit corresponds to logical low, while 1 corresponds to logical high. Only the logical high and low values may be assigned.

The number of bits in a binary, octal, or hexadecimal number must 'match' the number of signals to be assigned. This means that exactly the minimum number of data bits needed to assign a value to every signal must be given.

Assignment of a value to a bus is not recommended. Assignment of decimal values to signal lists longer than 32 bits is not supported. Assignment of values other than logical low and high is not recommended.

Trap Statement

Trap sets halting conditions for runs of the current generated module. A **run** of the module will stop (as if a **pause** had been issued) if the named signal changes to the specified value. Traps remain in place until they are taken out with **untrap**.

trap_statement :=
    trap command_signal_list = numeric_token ;

The command signal list has been described under command naming.

The numeric token (described under assignment) must match the length of the command signal list.

Untrap Statement

Untrap resets halting conditions previously entered by **trap**. Each trap must be removed

explicitly by stating the signal/value pair.

untrap_statement :=
    untrap command_signal_list = numeric_token ;

The command signal list has been described under command naming.

The numeric token (described under assignment) must match the length of the command signal list.

Show Statement

Show prints the value of a list of signals in a generated module. Each signal is printed individually, giving the signal value and time of last change.

A signal may have the following values:

- `0` logical low
- `1` logical high
- `U` undefined
- `X` bad signal value
- `T` tri-state value

show_statement :=
    show command_signal_list ;

Show works differently when used with a bus. If a primitive output port onto the bus is named, the value of the port is given, otherwise the computed bus value is printed. This enables all inputs to a bus to be printed, as well as the bus value.

Showvector Statement

Showvector prints a numeric equivalent of the signal values for a list of signals. The list of values is interpreted as a binary number, with the least significant bit corresponding to the rightmost element. Logical low corressponds to a 0 bit, while logical high corresponds to a 1. This is consistent with **assignment** rules for signals. The resulting composite value is printed as a decimal number. If any signal in the list has an abnormal value (not logical low or high), the signals are printed individually using **show**.

showvector_statement :=
    showvector command_signal_list ;

Showvector does not support signal lists containing more than 32 elements.

Showmessage Statement

Showmessage prints a message to simulator output. It takes a single string token and is designed to show progress through a command file being read.

showvector_statement :=
    showvector string_token ;

Quoted strings may be used.

Showtime Statement

Showtime prints the current simulation time.

showtime_statement :=
    showtime ;

Add Statement

Add appends signals to the show list of the current generated module. The list is initially empty and can be printed using **showlist**.

add_statement :=
    show command_signal_list ;

There is no way to delete a signal from the show list once it has been added.

Showlist Statement

Showlist prints (using **show**) the value of all signals in the show list of the current module. The list is maintained using the **add** command.

showlist_statement :=
    showlist ;

Display Statement

Display sets the display flags of a list of signals. A signal with its display flag set prints its state whenever it changes value. The flag is initially reset for all signals.

display_statement :=
    display command_signal_list ;

Display statement flags are independent of the **set display** flag.

Undisplay Statement

Undisplay resets the display flags of a list of signals. The flag is intially reset for all signals, and may be set using the **display** statement.

undisplay_statement :=
    undisplay command_signal_list ;

Display statement flags are independent of the **set display** flag.

Timed Statement

Timed statements store commands in the event queue of a generated module for execution during a run. Only **assignment** and **pause** statements may be used as timed statements. Timed statements have an initial argument which is the amount of simulation time to pass before the statement is executed. They are put into the event queue for execution.

timed_statement :=
    numeric_token : assignment_statement
    | numeric_token : pause_statement

pause functions differently when used as a timed statement. A timed **pause** halts the current simulation run, returning control to the command level which initiated the run (not necessarily the level which inserted the **pause**). This is similar to halting a run via an interrupt. The **assignment** statement functions normally.

Timed statements compute their target signals before being entered in the event queue. The present value of the current submodule is used for decoding '@'.

Quit Statement

Quit causes normal termination of the design tool. No state is retained between executions.

quit_statement :=
    quit ;

Chapter 4

Implementation Details

A simulation run iteratively executes primitive modules affected by changes to their input signals, then updates the value of their output signals. This continues until the simulation instance reaches a steady state, or a halt command is processed.

Each event in a simulation instance has an associated integer *processing time.* Events with the same processing time are completed in a single time step, and are processed before any event with a greater processing time. The last processing time executed is known as the *current processing time.* Simulating in time steps allows the current processing time to serve as an indicator of the amount of time a circuit takes to execute.

Following are specific implementation details of the design tool:

4.1 Primitive Modules

Primitive modules perform functions predetermined by C code. These modules have a uniform delay characteristic $\delta \geq 1$, meaning that a change on any of its inputs causes a change in its outputs exactly δ time units in the future.

The delay characteristic must be positive to satisfy the processing time requirement.

Uniformity ensures consistency in the output of a primitive module. Uniformity is needed because the simulation model does not throw out events. If the delay characteristic was nonuniform, a single module could cause schedule signal value changes on the same wire out of order.

4.2 Simulation Construction

To speed simulation, generated modules are flattened. *Flattening* removes the definition hierarchy from a simulation instance. Only instances of primitive modules and connections between them remain after flattening. This speeds execution, since the definition hierarchy

is not traversed during simulation. Flattening constructs connection lists for each signal that specify which primitive instances affect it and are affected by it.

The definition hierarchy is retained and is used to reference the flattened structure.

4.3 Bus Signals

Bus signals, which are driven by tri-state drivers, are built in a special way. Each primitive module on a bus writes to a specific entry point, similar to a port of a module. The bus value is calculated based on the values of its entry points. Every primitive module reading from the bus gets the calculated bus value.

Busses also have special handling for printing. A bus name which corresponds to an output of a primitive module prints information about the corresponding entry point. Any other name corresponding to the bus prints information about the calculated bus value. This allows for easier examination of busses.

4.4 Simulation Events

In order to satisfy the processing time requirement, events are stored in and read from a priority queue. This is implemented in the design tool by a heap.

The queue contains three types of events: signal value, printing and halting.

- Signal value events specify changes in the value of a signal. These events cause affected primitive modules to be executed.
- Printing events cause printing of signal information.
- Halting events stop execution of a simulation run following the current time step, instead of waiting until stable state.

Events may be created by a command statement, or as an effect of executing a primitive module.

4.5 Simulation Runs

Each simulation run reads and processes events until all events have been processed or a halting command has been processed.

Each time step of the run is conducted in phases.

1. All current events are extracted from the priority queue.

- Signal value events cause the target signal to be immediately updated. Each update causes connected busses and primitive modules to be scheduled for evaluation. Modules and busses are kept in separate evaluation lists. If a signal is updated more than once in a single time unit, an error message is printed.
- Printing events get stored in a list for later processing.
- A halting command sets a flag to exit the simulation run following the current time unit.

2. Busses scheduled in the first phase are evaluated, based on the value of all signals connected to it. This may cause schedule additional primitive modules for evaluation.

3. Each primitive module in the evaluation list is processed. The C code for each affected module is executed. This may change internal state and may schedule additional simulation events. Because of the delay characteristic of primitive modules, events are always scheduled for a later processing time.

4. Printing commands are executed. This shows the signal state at the end of the current processing time.

After these phases are completed, the simulation stops if the halting flag is set. Otherwise, the next time step is processed.

Chapter 5

Startup Options

The design tool has a number of options which are set at the beginning of execution. These are separate from command statements and do not change during execution. The options control general input and output characteristics of the design tool.

Commands are normally read from `stdin` and output written to `stdout`. Error messages are directed to `stderr`. Input and output may be redirected by using startup options. Error messages may not be redirected.

5.1 Command-line Arguments

A number of options may be set upon execution of the design tool.

sim [option_list]

Acceptable command line options are:

-i <filename> Read commands from the named file instead of `stdin`. This causes batch mode, instead of interactive, execution.

-o <filename> Direct output messages to the named file instead of `stdout`. An output file should be specified only when in batch mode (**-i**).

-s Print information on all signals when generating modules. These messages are useful in circuit design verification.

-ns Only print information about signals which have bad drive/load ratios. This is the opposite of **-s**.

-b Run code to inspect bus loads. The design tool passes load through transmission gates to give a different description of the load on module outputs.

-nb Do not inspect bus loads. This is the opposite of **-b**.

5.2 simrc File

Upon startup, information is read from a file named **simrc**, which should be in the current directory upon program execution. **simrc** is read before the command line arguments are interpreted, so it can be used to change default switch values for signal printing and bus load computation.

Comments in **simrc** are specified by the pound sign '**#**', similar to other syntax rules. Numbers in **simrc** are interpreted using the C "strtol" function. These do not conform to the conventions used in other parts of the design tool.

There are four options which may be specified in **simrc**:

Signal Printing

Information on all signals are printed when the keyword **print_signals** is specified. This is the same as using the **-s** command-line argument.

Bus Load

Additional bus load computation may be selected by using the keyword **print_busses**. This is the same as using the **-b** command-line argument.

Fanout

Fanout is a crude measure of the drive/load ratio on signals. Signals with large numbers of inputs or outputs are more likely to have load problems. A rough estimate of signal load is produced by comparing the number of inputs and outputs of each signal to a user-specified number. A warning message is printed for each signal which has a fan-in or fan-out greater than the fanout value.

The fanout value is specified with the keyword **max_fan**, followed by an integer. The number should use C syntax.

Error Printing

Error messages may be supressed by specification of an error printing mask. Only errors specified by the mask get printed. The list of error types and their corresponding mask numbers are shown in appendix D.

The error printing mask is specified with the keyword **print_mask**, followed by an integer. The number should use C syntax.

Error Halting

Specified errors can force termination of the design tool by use of an error halting mask. Encountering an error specified by the mask causes the design tool to exit. The list of error types and their corresponding mask numbers are shown in the appendix D.

The error halting mask is specified with the keyword **halt_mask**, followed by an integer. The number should use C syntax.

Errors specified by the halting mask always print before exiting, even when not specified by the printing mask.

Appendix A

The Lexer

A lexer is used to convert input into tokens.

The lexer recognizes four primary types of tokens:

- single character tokens
- string tokens
- reserved words
- numeric tokens

The lexer uses spacing characters (space, tab, newline) to separate tokens, but they are not passed along.

Comments

The character '#' is used to signify a comment. When a comment character is read, the remainder of the input line (until the next newline) is disregarded. Commenting does not work within a quoted string.

Single Character Tokens

The single characters tokens recognized by the lexer are:

',', ':', ';', '=', '!', '<', '>', '(',
')', '[', ']', '{', '}', '-', '+', '*',
'/', '%', '.', '@', '^', '|', '&'

Single character tokens do not need to be separated from other tokens by spacing characters.

Non-alphanumeric characters which are not single character tokens or one of the special characters '_', '#', and '"' are disregarded.

String Tokens

The lexer recognizes two types of string tokens: quoted and unquoted.

An *unquoted string* consists of an initial alphabetic character or underscore '_' followed by any number of alphanumeric characters or underscores. Unquoted strings are checked against the list of reserved words. If an unquoted string matches a reserved word, it is passed to the simulator as the reserved word token.

A *quoted string* is a succesion of characters enclosed within two delimiting quote symbols '"'. Quoted strings allow acceptance of strings which do not qualify as unquoted strings. This is used for filenames and message printing. A quoted string may not cross a line boundary. Quoted strings are not checked against reserved words, so they are always passed as string tokens.

Here is the string syntax given as regular expressions:

```
unquoted_string   := [a-zA-Z_][a-zA-Z_0-9]*
quoted_string     := "?*"
```

In the regular expressions, square brackets denote a choice between characters. '?' represents any single character. '*' means a sequence of zero or more of the previous character or choice of characters.

There is currently no way to pass a string containing the newline character.

Reserved Words

Reserved words are strings which have special meaning in the design tool. Each unquoted string read by the lexer is checked against the list of reserved words. If a string matches a reserved word, it is passed as the reserved word.

There are two categories of reserved words. The first is used when reading definitions, the other when reading commands.

Reserved Definition Words:

break	cable	cmos	components	cost
else	end	error	for	gateInputs
if	inout	input	join	module
nmos	output	ports	signals	ts_inout
ts_output	while			

Reserved Command Words:

HSIG	LSIG	TSIG	USIG	XSIG
add	brief	clear	close	destroy
display	generate	load	pause	read
repeat	reset	run	save	set
show	showlist	showmessage	showtime	showvector
simulation	source	trap	undisplay	until
untrap	verbose	while		

Numeric Tokens

Four types of numeric tokens are recognized by the lexer: binary, octal, decimal, and hexadecimal. These correspond to numbers in base 2, 8, 10, and 16 respectively.

Binary, octal, and hexadecimal numbers have '0' as their initial character. The second character specifies the base of the number.

- 'b' or 'B' specifies a binary number. This is followed by a sequence of the characters '0' or '1'.
- 'o' or 'O' specifies an octal number. This is followed by a sequence which may contain characters corresponding to the numbers 0-8.
- 'x' or 'X' specifies a hexadecimal number. This is followed by a sequence which may contain characters corresponding to the numbers 0-9 or alphanumeric characters in the range a-f (upper or lower case). The characters a-f represent the decimal values 10-15 respectively.

If the second character does not fall into the above categories or if the leading character is a number which is not '0', the numeric token is a decimal number. A decimal number is a sequence of characters, each of which corresponds to a number in the range 0-9.

Each syntax is repeated below as a regular expression.

```
binary_number      := 0[bB][01]*
octal_number       := 0[oO][0-8]*
hexadecimal_number := 0[xX][0-9a-fA-F]*
decimal_number     := [0-9][0-9]*
```

In the regular expressions, square brackets denote a choice between characters. '*' means a sequence of zero or more of the previous character or choice of characters.

All types of numeric tokens are interpreted as having the most significant digit on the left.

Appendix B

Primitive Modules

This appendix contains the current list of predefined primitive modules. Each primitive is shown as a module definition, with associated costs, and is accompanied by a short description.

Improper input values to primitive modules cause uncertain results to occur. These results should not be relied upon. The following rules generally apply:

- **bad** signal values propagate.
- If no **bad** signals are present, **undefined** signals propagate.
- If no **bad** signals are present, **tri-state** signals cause **undefined** output.

Because primitive modules are specially defined, some of their functions cannot be reproduced by general composite modules.

Constant

const allows signals to be hooked to a constant source. The input argument becomes the source value. Valid argument values are '0' (logical low) and '1' (logical high). Use of other values is not recommended.

Constant values cause attached modules to execute on the first run following module generation and after a simulation instance has been reset.

```
module const(v)
# the constant has zero costs
cost nmos: 0  cmos: 0  gateInputs: 0
   ports
    v output
end
```

Inverter

inv does a logical inversion of the input signal. Valid input values for "a" are logical low and high.

- If "a" is low, "x" is set to high.
- If "a" is high, "x" is set to low.

```
module inv
cost nmos: 2  cmos: 2  gateInputs: 1
  ports
    a input
    x output
end
```

Logical NAND

nand computes the logical NAND of the input signals. Valid input values for the inputs "a" and "b" are logical low and high.

- If either signal is low, "x" is set to high.
- If both signals are high, "x" is set to low.

```
module nand
cost nmos: 3  cmos: 4  gateInputs: 2
  ports
    a input
    b input
    x output
end
```

Logical NOR

nor computes the logical NOR of the input signals. Valid input values for the inputs "a" and "b" are logical low and high.

- If either signal is high, "x" is set to low.
- If both signals are low, "x" is set to high.

```
module nor
cost nmos: 3  cmos: 4  gateInputs: 2
  ports
    a input
    b input
    x output
end
```

Delay

delay simply propagates a signal value with a time delay. The output signal is set to the input signal, regardless of the value. The input argument is the time to delay the output, which must be a positive number.

The cost of a **delay** is represented as a pair of inverters.

```
module delay(delta)
cost nmos: 4  cmos: 4  gateInputs: 1
  ports
    d input
    q output
end
```

Transmission Gate

trans_gate sets the output "q" to the value of the input "d" when enabled with the enable signals "e1" and "e2". When not enabled, "q" is set to the tri-state value. The transmission gate is a dual-rail model, which means "e1" should always be the logical inverse of "e2".

- When "e1" is high ("e2" is low), "q" gets the value of "d".
- When "e1" is low ("e2" is high), "q" gets the tri-state value.

"q" must be hooked to a bus signal. This means that all ports which output to the bus must be typed as tri-state. In particular, only **trans_gate** outputs and **SRAM** data lines may output to the same signal as "q".

```
module trans_gate
cost nmos: 1  cmos: 2  gateInputs: 2
  ports
    d input
    e1 input
    e2 input
    q ts_output
end
```

Positive Latch

posLatch is a single bit of non-volatile memory. It uses "l" to control when data is read into memory from "d". The value in memory is output through "Q".

- When "l" is low, the latch holds state.
- When "l" is high, the memory value (and "Q") is set to the value of "d".

```
module posLatch
cost nmos: 8  cmos: 10
  ports
    d input
    l input
    Q output
end
```

Negative Latch

negLatch is a single bit of non-volatile memory. It uses "l" to control when data is read into memory from "d". It is called a negative latch (as opposed to positive latch) because the sense of the latch signal "l" is reversed. The value in memory is output through "Q".

- When "l" is low, the memory value (and "Q") is set to the value of "d".
- When "l" is high, the latch holds state.

```
module negLatch
cost nmos: 8  cmos: 10
  ports
    d input
    lb input
    Q output
end
```

Fan_out

fan_out propagates a signal value with a time delay, similar to a delay module. The output signal is set to the input signal. The first argument is the number of inverters which can be driven by the output of the fan-out module. The second argument is the time the output is delayed after the input. Both arguments must be positive numbers.

fan_out is intended to be used in a composite design, along with externally specified costs, to simplify simulation of a fan-out structure.

```
module fan_out(no, delta)
cost nmos: 0  cmos: 0  gateInputs: 0
  ports
    i input
    0 output
end
```

Two_way_fan_out

two_way_fan_out, like **fan_out**, is used to drive multiple outputs from a single signal. This design allows the use of both the signal and its inverse as drivers. The first argument is the number of inverters which can be driven by the non-inverted output line of the module. The second argument is the number of inverters which can be driven by the inverted output line of the fan-out module. The last argument is the time the outputs are delayed after the input. Both outputs use the same delay characteristic. All arguments must be positive numbers.

two_way_fan_out is intended to be used in a composite design, along with externally specified costs, to simplify simulation of a fan-out structure.

```
module two_way_fan_out(nuo, nio, delta)
cost nmos: 0  cmos: 0  gateInputs: 0
  ports
    i input
    0 output
    0_b output
end
```

Static RAM

SRAM is memory for simulation instances. A static RAM module takes two arguments: the amount of memory and the number of bits in the word. It reads and stores data in addressable memory based on its control signals "rw" and "e". "e" enables the RAM for an operation, and "rw" selects whether the operation reads from or writes to memory.

- If "e" is low, the memory does nothing.
- If "e" is high, the memory does the specified operation.
- If "rw" is low, the operation is a write.
- If "rw" is high, the operation is a read.

The signals in "D" must be hooked to busses. This means all ports which output to each bus must be typed as tri-state. In particular, only **trans_gate** outputs and other **SRAM** data lines may output to those busses.

There is currently no way to preload data into the memory. All data must be written to memory before it is used.

Data widths larger than 32 bits are not supported.

```
module SRAM(amount, width)
  ports
    rw input
    e  input
    A[1:amount] input
    D[1:width]  ts_inout
end
```

Dynamic Memory Test

D_test is used to simulate dynamic RAM in conjunction with the static RAM module **SRAM**. It keeps track of the last time data was written to the address, however **D_test** does not actually store the data. If data is used too long after it has last been written, an error message is generated.

The "rw" and "e" lines work as described for the static RAM.

```
module D_test(amount)
  ports
    rw input
    e  input
    A[1:amount] input
end
```

Appendix C

Module Examples

This section contains two simple examples which demonstrate certain features in the definition language. The examples have not been optimized.

The first example is a scalable multi-input OR. It is constructed using a two-input OR, which in turn is built from primitives **nor** (logical NOR) and **inv** (inverter). The multi-input OR uses recursion to construct a collection tree. This results in an $O(log(k))$ running time as opposed to $O(k)$ time for a chain.

```
module two_input_OR
  ports
    x y input
    z   output
  signals
    z_bar
  components
    xy_nor nor  x y    z_bar;
    z_comp inv  z_bar  z;
end

module k_input_OR(k)
# compute a multi-input OR by recursion
  ports
    x[1:k] input
    z      output
  signals
    z1 z2 # internal signals for split
  components
    if {k = 1} {
      join [ x[1] z ];  # connect input to output
      break (1);        # end current module; halt recursion
    }
```

```
# compute split information
    k1 <- k/2;
    k2 <- k - k1;

# split in half and recurse on both parts.
    z1_comp k_input_OR(k1)  x[1:k1]    z1;
    z2_comp k_input_OR(k2)  x[k1+1:k]  z2;

# recombine parts
    z_comp two_input_OR  z1 z2  z;
end
```

This example uses recursion to split the tree into two subtrees, and a **two_input_OR** to recombine the subtrees. Recursion is halted when the subtree has only a single input. This is done by using **break** to end the module definition.

The second example is a variable length MIN circuit. It uses a for loop to join a chain of single-bit MIN modules.

```
module two_input_AND
  ports
    x y input
    z   output
  signals
    z_bar
  components
    xy_nand nand  x y    z_bar;
    z_comp  inv   z_bar  z;
end

module a_gre_b
# set z to one if a is greater than b ((a = 1) & (b = 0))
  ports
    a b input
    z   output
  signals
    b_bar
  components
    b_inv  inv  b  b_bar;
    z_comp two_input_AND  a b_bar  z;
end

module MIN
```

```
# compute MIN based on input values and selection inputs
# compute selection outputs for chaining
# (sxi = 1) -> choose x as the MIN
# (syi = 1) -> choose y as the MIN
# sxi = syi = 1 is an impossible state
  ports
    x y input
    z   output
    sxi syi input   # select control input
    sxo syo output  # select control output
  signals
    z1 z2 z3
    sxi_bar syi_bar
    x_gre_y y_gre_x  # used to find out which data input is greater
    sxo_e syo_e      # contains new select information
  components
# compute the min value z
    x_sel  two_input_AND  x sxi  z1;
    y_sel  two_input_AND  y syi  z2;
    xy_and two_input_AND  x y    z3;
    z_comp k_input_OR(3)  z1 z2 z3  z;


    sxi_inv inv  sxi  sxi_bar;
    syi_inv inv  syi  syi_bar;

# check if values are not equal
    x_gre_y_comp a_gre_b  x y  x_gre_y;
    y_gre_x_comp a_gre_b  y x  y_gre_x;

# compute new select information
    sxo_e_comp two_input_AND  y_gre_x syi_bar  sxo_e;
    syo_e_comp two_input_AND  x_gre_y sxi_bar  syo_e;

# compute output selects
    sxo_comp two_input_OR  sxi sxo_e  sxo;
    syo_comp two_input_OR  syi syo_e  syo;
end
```

MIN uses information from the input select lines or by comparing the two signals x and y to compute the output z and the output select lines. Note that reversing the order of signals connected to the ports of the circuit **a_gre_b** changes its function.

```
module k_bit_MIN(k)
# compute a variable length MIN circuit by iteration of
```

```
# a chainable single bit MIN
  ports
    x[1:k] y[1:k] input
    z[1:k] output
  signals
    sx[0:k] sy[0:k] low
  components
# Turn off initial select signals
    low_gen const(0)  low;
    join [ low sx[0] sy[0] ];

# Chain MIN circuits together
    for i = 1,k
      bit[i] MIN  x[i] y[i]  z[i]  sx[i-1] sy[i-1]  sx[i] sy[i];
end
```

The chain is initialized by connecting the first set of select inputs to the **low** signal. The last set of select outputs is left unconnected.

Appendix D

Error Messages

Error messages each have an associated field which describes its type. The error type is used to identify groups of messages for special consideration. Upon startup, a print mask and a halt mask are read from the **simrc** file. The print mask specifies error types which are printed. The halt mask specifies error types which halt design tool execution. Each message that halts execution is automatically printed.

Following is a list of error masks and their associated groupings. Each mask is given as an octal constant.

000001L	Race condition during a simulation run
000002L	Corrected parsing error
000004L	Warning
000010L	Redefinition of a cable or module
000020L	Reference to undefined cable or module
000040L	Conflicting definitions
000100L	Uncorrectable parsing error
000200L	Module generation halted
000400L	Error in primitive module
001000L	Bad data found
002000L	**Error** statement executed
004000L	Memory allocation error
010000L	Error external to program
020000L	Inconsistency in program